TDD Changed My Life

It's 7:15 am and customer support is swamped. We just got featured on Good Morning America, and a whole bunch of first time customers are bumping into bugs.

It's all-hands-on-deck. We're going to ship a hot fix NOW before the opportunity to convert more new users is gone. One of the developers has implemented a change he thinks will fix the issue. We paste the staging link in company chat and ask everybody to go test the fix before we push it live to production. It works!

Translation: 所有人都在甲板上。在转换更多新用户的机会消失之前，我们将立即推出热修复程序。其中一位开发人员已经实施了一项更改，他认为这将解决问题。我们在公司聊天中粘贴阶段链接，并要求每个人在我们将其推送至生产之前先测试修复程序。它有效！

Our ops superhero fires up his deploy scripts, and minutes later, the change is live. Suddenly, customer support call volume doubles. Our hot-fix broke something else, and the developers erupt in synchronized git blame while the ops hero reverts the change.

Translation: 我们的超级英雄启动了他的部署脚本，几分钟后，变化就开始了。突然间，客户支持电话量翻了一番。我们的热修复破坏了其他东西，当操作英雄恢复更改时，开发人员爆发出同步的git指责。

Why TDD?

It's been a while since I've had to deal with that situation. Not because developers stopped making mistakes, but because for years now, every team I've led and worked on has had a policy of using TDD. Bugs still happen, of course, but the release of show-stopping bugs to production has dropped to near zero, even though the rate of software change and upgrade maintenance burden has increased exponentially since then.

Whenever somebody asks me why they should bother with TDD, I'm reminded of this story — and dozens more like it. One of the primary reasons I switched to TDD is for improved test coverage, which leads to 40%-80% fewer bugs in production. This is my favorite benefit of TDD. It's like a giant weight lifting off your shoulders.

TDD eradicates fear of change.

On my projects, our suites of automated unit and functional tests prevent disastrous breaking changes from happening on a near-daily basis. For example, I'm currently looking at 10 automated library upgrades from the past week that I used to be paranoid about merging because what if it broke something?

Translation: 在我的项目中，我们的自动化单元和功能测试套件可以防止灾难性的破坏性更改几乎每天都会发生。例如，我目前正在研究过去一周的10个自动库升级，我过去对合并很偏执，因为如果它坏了怎么办？

All of those upgrades integrated automatically, and they're already live in production. I didn't look at a single one of them manually, and I'm not worried at all about them. I didn't have to go hunting to come up with this example. I popped open GitHub, looked at recent merges, and there they were. What was once manual maintenance (or worse, neglect) is now automated background process. You could try that without good test coverage, but I wouldn't recommend it.

What is TDD?

TDD stands for Test Driven Development. The process is simple:

Red, Green, Refactor

Before you write implementation code, write some code that proves that the implementation works or fails. Watch the test fail before moving to the next step (this is how we know that a passing test is not a false positive — how we test our tests).

Translation: 在编写实现代码之前，请编写一些代码来证明实现是有效的还是失败的。在进入下一步之前，请注意测试失败（这就是我们如何知道通过测试不是假阳性——我们如何测试测试）。

Write the implementation code and watch the test pass.

Translation: 编写实现代码并观察测试通过。

Refactor if needed. You should feel confident refactoring your code now that you have a test to tell you if you've broken something.

Translation: 如果需要，请重新调整。既然有测试可以告诉你是否破坏了某些东西，那么你应该对重构代码充满信心。

How TDD Can Save You Development Time

On the surface, it may seem that writing all those tests is a lot of extra code, and all that extra code takes extra time. At first, this was true for me, as I struggled to understand how to write testable code in the first place, and struggled to understand how to add tests to code that was already written.

TDD has a learning curve, and while you're climbing that learning curve, it can and frequently does add 15% — 35% to implementation times. But somewhere around the 2-years in mark, something magical started to happen: I started coding faster with unit tests than I ever did without them.

Translation: TDD有一个学习曲线，当你在攀登这个学习曲线时，它可以而且经常会增加15%-35%的实现时间。但在2年左右的某个时候，神奇的事情开始发生：我开始用单元测试比没有单元测试更快地编码。

Several years ago I was building a video clip range feature in a UI. The idea was that you'd set a starting point and an ending point for a video, and when the user links to it, it would link to that precise clip rather than the whole video.

Translation: 几年前，我正在UI中构建一个视频剪辑范围功能。这个想法是，你可以为视频设置一个起点和终点，当用户链接到它时，它会链接到精确的剪辑，而不是整个视频。

But it wasn't working. The player would reach the end of the clip and keep on playing, and I had no idea why.

I kept thinking it had to do with the event listener not getting hooked up properly. My code look something like this:

```
video.addEventListener('timeupdate', () => {
  if (video.currentTime >= clip.stopTime) {
    video.pause();
  }
});
```

Change. Compile. Reload. Click. Wait. Repeat.

Each change took almost a minute to test, and I tried a hilariously large number of things (most of them 2–3 times).

Translation: 每一个变化都花了将近一分钟的时间来测试，我尝试了很多有趣的东西（大多数都是2-3次）。

Did I misspell timeupdate? Did I get the API right? Is the video.pause() call working? I'd make a change, add a console.log(), jump back into the browser, hit refresh, click to a moment before the end of the clip, and then wait patiently for it to hit the end. Logging inside the if statement did nothing. OK, that’s a clue. Copy and paste timeupdate from the API docs to be absolutely sure it wasn’t a typo. Refresh, click, wait. No luck!

Translation: 我拼错时间更新了吗？API我说得对吗？video.pause（）调用有效吗？我会做一个改变，添加一个console.log（），跳回浏览器，点击刷新，点击剪辑结束前的一刻，然后耐心等待剪辑结束。在if语句中登录没有任何作用。好吧，这是个线索。复制并粘贴API文档中的时间更新，以确保它不是打字错误。刷新，单击，等待。运气不好！

Finally, I placed a console.log() outside the if statement. “This can’t help,” I thought. After all, that if statement was so simple, there’s no way I could have screwed up the logic. It logged. I spit my coffee on the keyboard. WTF?!

Murphy’s Law of Debugging: The thing you believe so deeply can’t possibly be wrong so you never bother testing it is definitely where you’ll find the bug after you pound your head on your desk and change it only because you’ve tried everything else you can possibly think of.

I set a breakpoint to figure out what was going on. I inspected the value of clip.stopTime. undefined??? I looked back at my code. When the user clicks to select the stop time, it places the little stop cursor icon, but never sets clip.stopTime. “OMG I’m a gigantic idiot and nobody should ever let me anywhere near a computer again for as long as I live.”

Years later I still remember this because of that feeling. You know exactly what I’m talking about. We’ve all been there. We’re all living memes.

Actual photos of me while I’m coding.

Translation: 我编码时的实际照片。

If I was writing that UI today, I’d start with something like this:

Translation: 如果我今天在写UI，我会从以下内容开始：

```
describe('clipReducer/setClipStopTime', async assert => {
  const stopTime = 5;
  const clipState = {
    startTime: 2,
    stopTime: Infinity
  };

  assert({
    given: 'clip stop time',
    should: 'set clip stop time in state',
    actual: clipReducer(clipState, setClipStopTime(stopTime)),
    expected: { ...clipState, stopTime }
  });
});
```

Granted, superficially, that looks like a whole lot more code than clip.stopTime = video.currentTime. But that's the point. This code acts like a specification. Documentation, along with proof that the code works as documented. And because it exists, if I change the way I position the stop time cursor in the UI, I don't have to worry about whether or not I'm breaking the clip stop time code in the process.

Translation: 当然，从表面上看，这看起来比clip.stopTime=video.currentTime要多得多。但这就是重点。此代码的作用类似于规范。文档，以及代码按文档工作的证明。因为它的存在，如果我改变在UI中定位停止时间光标的方式，我就不必担心在这个过程中是否破坏了剪辑停止时间代码。

Note: Want to write unit tests like this? Check out "Rethinking Unit Test Assertions".

The point is not how long it takes to type this code. The point is how long it takes to debug it if something goes wrong. If this code broke, this test would give me a great bug report. I'd know right away that the problem is not the event handler. I'd know it's in setClipStopTime() or the clipReducer() which implements the state mutation. I'd know what it's supposed to do, the actual output, and the expected output — and more importantly — so would a coworker, 6-months into the future who's trying to add features to the code I built.

One of the first things I do in every project is set up a watch script that automatically runs my unit tests on every file change. I often code with two monitors side-by-side and keep my dev console with the watch script running on one monitor while I code on the other. When I make a change, I usually know within 3 seconds whether or not that change worked.

For me, TDD is far more than a safety net. It's also constant, fast, realtime feedback. Instant gratification when I get it right. Instant, descriptive bug report when I get it wrong.

TDD Taught Me How to Write Better Code

I'm going to admit something embarrassing: I had no idea how to build apps before I learned TDD with unit testing. How I ever got hired would be beyond me, but after interviewing hundreds and hundreds of developers, I can tell you with great confidence: there are a lot of developers in the same boat. TDD taught me almost everything I know about effective decoupling and composition of software components (meaning modules, functions, objects, UI components, etc.)

Translation: 我要承认一件尴尬的事情：在我学习TDD和单元测试之前，我不知道如何构建应用程序。我不知道我是如何被录用的，但在面试了成百上千的开发人员后，我可以非常自信地告诉你：有很多开发人员在同一条船上。TDD教会了我几乎所有关于软件组件（即模块、函数、对象、UI组件等）的有效解耦和组合的知识

The reason for that is because unit tests force you to test components in isolation from each other, and from I/O. Given some input, the unit under test should produce some known output. If it doesn't, the test fails. If it does, it passes. The key is that it should do so independent of the rest of the application. If you're testing state logic, you should be able to test it without rendering anything to the screen or saving anything to a database. If you're testing UI rendering, you should be able to test it without loading the page in a browser or hitting the network.

Among other things, TDD taught me that life gets a lot simpler when you keep UI components as minimal as you can. Isolate business logic and side-effects from UI. In practical terms, that means that if you're using a component-based UI framework like React or Angular, it may be advantageous to create display components and container components, and keep them separate.

For display components, given some props, always render the same state. Those components can be easily unit tested to be sure that props are correctly wired up, and that any conditional logic in the UI layout works correctly (for example, maybe a list component shouldn't render at all if the list is empty, and it should instead render an invitation to add some things to the list).

I knew about separation of concerns long before I learned TDD, but I didn't know how to separate concerns.

Unit testing taught me about using mocks to test things, and then it taught me that mocking is a code smell, and that blew my mind and completely changed how I approach software composition.

Translation: 单元测试教会了我如何使用mock来测试东西，然后它教会了我mock是一种代码气味，这让我大吃一惊，彻底改变了我处理软件组成的方式。

All software development is composition: the process of breaking large problems down into lots of small, easy-to-solve problems, and then composing solutions to those problems to form the application. Mocking for the sake of unit tests is an indication that your atomic units of composition are not really atomic, and learning how to eradicate mocks without sacrificing test coverage taught me how to spot a myriad of sneaky sources of tight coupling.

That has made me a much better developer, and taught me how to write much simpler code that is easier to extend, maintain, and scale, both in complexity, and across large distributed systems like cloud infrastructure.

How TDD Saves Whole Teams Time

I mentioned before that testing first leads to improved test coverage. The reason for that is that we don't start writing the implementation code until we've written a test to ensure that it works. First, write the test. Then watch it fail. Then write the implementation code. Fail, pass, refactor, repeat.

That process builds a safety net that few bugs will slip through, and that safety net has a magical impact on the whole team. It eliminates fear of the merge button.

That reassuring coverage number gives your whole team the confidence to stop gatekeeping every little change to the codebase and let changes thrive.

Translation: 这个令人放心的覆盖率数字让您的整个团队有信心停止对代码库的每一个微小更改进行把关，让更改蓬勃发展。

Removing fear of change is like oiling a machine. If you don't do it, the machine grinds to a halt until you clean it up and crank it, squeaking and grinding back into motion.

Without that fear, the development cadence runs a lot smoother. Pull requests stop backing up. Your CI/CD is running your tests — it will halt if your tests fail. It will fail loudly, and point out what went wrong when it does.

And that has made all the difference.

Want to Learn More About TDD?

TDD Day is an all-day recorded webinar. Members of EricElliottJS.com can watch the recording.

Translation: TDD日是一个全天录制的网络研讨会。EricElliott JS.com的成员可以观看录音。

You'll learn:

Why TDD has taken over

Economics of software quality

Unit vs functional vs integration tests

5 questions every unit test must answer

Translation: 每个单元测试必须回答5个问题

TDD the RITE way

Translation: TDD的RITE方式

Mocking is a code smell

Why testable software leads to better architecture

Causes of tight coupling

How to do more with pure functions

Translation: 如何用纯函数做更多的事情

Unit testing React components

Translation: 单元测试React组件

Sign up to get started.

Eric Elliott is a tech product and platform advisor, author of “Composing Software”, cofounder of EricElliottJS.com and DevAnywhere.io, and dev team mentor. He has contributed to software experiences for Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC, and top recording artists including Usher, Frank Ocean, Metallica, and many more.

He enjoys a remote lifestyle with the most beautiful woman in the world.